Behavioral Modeling and Run-Time Verification of System-of-Systems Architectural Requirements

Doron Drusinsky, J. Bret Michael and Man-Tak Shing
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943, USA
{ddrusins, bmichael, shing}@nps.edu

ABSTRACT

This paper addresses the need to analyze and validate the architectural requirements of complex systems-of-systems early in the development process. We present an iterative rapid prototyping process for studying the temporal behavior of the system-of-systems architecture via object-oriented architectural models expressed in UML-RT (UML for Real-Time extension) augmented with formal specification of timing requirements in terms of a time-series temporal logic. The UML-RT models are translated into coarse-grained simulation models that are exercised using the OMNeT++ simulation engine. We instrument the OMNeT++ simulation code with probes that send information to DBRover, a time-series temporal logic tool that simulates, builds, executes and monitors temporal assertions about a target application, for temporal property verification during prototype execution. We illustrate the proposed approach with a case study of the sensor-netting capability of a missile defense system.

Keywords: UML-RT, Real-time System, Temporal Logic, Run-time Execution Monitoring, Execution-based Model Checking, Missile Defense

1. INTRODUCTION

The analysis and architectural design of complex systems-of-systems, such as the Ballistic Missile Defense System, pose many challenges [2,3]. First, the system is complex and yet has to be highly dependable. In addition, these systems are often distributed, heterogeneous, network-centric, and software intensive. Any good architecture for such systems must be easily evolvable and reconfigurable since it has to accommodate legacy systems as well as systems under development. These systems have to operate in unpredictable environments and little is known about how to model and reasoning about such complex systems. Feasible requirements for these systems are difficult to formulate, understand, and meet without extensive prototyping. Modeling and simulation holds the key to the rapid construction and evaluation of prototypes early in the development process. Moreover, we need a process that can be readily adaptable by the defense industry.

There is a growing interest in using object-oriented analysis and design techniques in conjunction with the Unified Modeling Language (UML) [13] to develop complex systems-of-systems. In [11], we presented an iterative process (Figure 1) for studying the temporal behavior of system-of-systems architecture via object-oriented architectural models expressed in UML-RT (UML for Real-Time extension) [15].

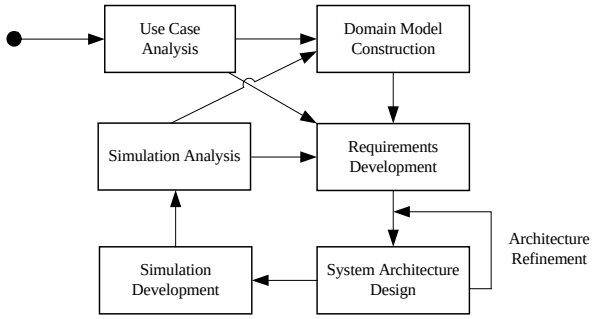

Figure 1. The Iterative Prototyping Process

The process begins with use case analysis to identify user needs. Based on the use cases, we develop an object-oriented distributed architecture of the system expressed in UML-RT. We refine the internal structures of the component systems using the Hierarchy plus Input, Process, Output (HIPO) technique [7] until components are readily mapped to modules of the target simulation written in OMNeT++ [17]. The UML-RT models are then translated into coarse-grained simulation models that are exercised using the OMNeT++ simulation engine. We use the simulation to study the feasibility and correctness of the timing requirements and apply the lessons learned to modify the system architecture and timing constraints accordingly.

The real-time nature of the missile defense requires the support of real-time systems whose correct behavior depends not only on the logical result of the computation but also on the time at which the result is produced. Traditionally, these temporal requirements are expressed as hard and soft timing constraints. It is imperative for real-time systems to meet all deadlines in hard timing constraints but acceptable to miss the deadlines of the soft timing constraints occasionally [8]. There are currently two complementary approaches to evaluating the correctness of real-time systems: static analysis of its behavior according to a set of metrics (e.g., schedulability analysis to establish the feasibility of the timing constraints) and run-time monitoring of real-time systems to study its behavior according to a set of metrics (e.g., release jitter, frequency and degree of tardiness). While the static analytic approach plays an important role in helping system designers set time budgets and allocate resources in their designs, they are only effective if correct timing constraints can be determined during the requirements analysis

phase. Moreover, traditional analytical techniques are not effective in evaluating time-series temporal behaviors (e.g., the average duration between consecutive missing deadlines within any 10-minute interval must be greater than 5 seconds). This kind of requirement can only be evaluated through execution of the real-time systems or their prototypes.

This paper describes the use of time-series temporal logic to capture the timing properties of a system-of-systems architecture. The temporal assertions are input to DBRover, a time-series temporal logic tool that builds and executes temporal rules for the target application [5], to generate executable timing specifications. We then instrumented the OMNeT++ simulation code with probes (code snippets) to send information to DBRover for temporal property verification during prototype execution. The rest of the paper is organized as follows. Section 2 provides an introduction to UML-RT. Section 3 gives an overview of temporal logic and the DBRover system. Section 4 presents a case study of a missile defense system to demonstrate the use of time-series temporal logic for the run-time verification of timing properties. Section 5 presents a discussion of the approach.

2. UML-RT

UML-RT is an extension of UML and is based on the concepts underlying the ROOM language [16]—an architectural definition language developed specifically for complex real-time software systems. UML-RT provides three principal constructs (*capsules*, *ports*, and *connectors*) for modeling the structures of a real-time system. *Capsules* are specialized UML active objects for modeling self-contained components of a system with the following two restrictions: (1) capsule operations can only be called within the capsule and (2) capsules can only communicate with other capsules through special mechanisms called ports. *Ports* are objects within a capsule that act as interfaces on the boundary of the capsule. A capsule may have one or more ports through which it is interconnected with other capsules via connectors. *Connectors* represent communication channels through which capsules communicate via the sending and receiving of messages. Each port is associated with a *protocol* that captures the semantics of the interactions between the port and its counterpart on the opposite end of the connector.

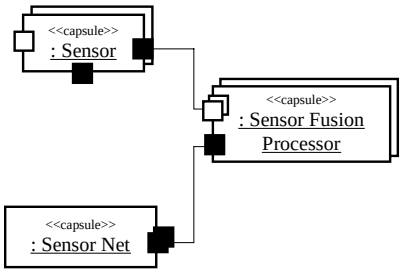

Figure 2. A UML-RT model

Figure 2 shows a simple UML-RT model consisting of a set of sensor capsules, a set of sensor fusion processor capsules and a sensor net capsule. Each sensor capsule has three ports. It uses one of the ports to communicate with its associated sensor fusion processor capsule. Each sensor fusion processor capsule has multiple ports for communication with its associated sensors (as indicated by the multi-object icon) and uses a single port to communicate with the sensor net capsule. The "white-filled" icons on the sensor fusion processor capsule indicate that the sensor fusion processor capsule plays the "slave" role of a binary protocol when communicating with its associated sensor capsules. A capsule may contain collaborating sub-capsules, as shown in Figure 3, and may have at most one state machine that specifies the dynamic behavior of the capsule.

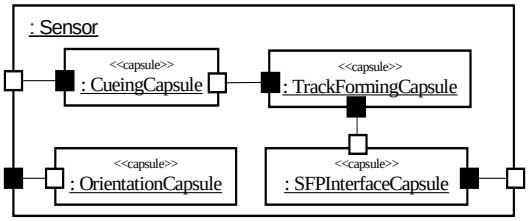

Figure 3. The internal view of the sensor capsule

3. METRIC TEMPORAL LOGIC WITH TIME SERIES CONSTRAINTS

Temporal Logic is a special branch of modal logic for investigating the notion of time and order. In [14], Pnueli suggested using Linear-Time Propositional Temporal Logic (LTL) for reasoning about concurrent programs. Since then, several researchers have used LTL to state and prove correctness of concurrent programs, protocols, and hardware (e.g., [6,9]). LTL is an extension of propositional logic where, in addition to the propositional logic operators there are four future-time operators and four dual past-time operators: a*lways* in the future (*always* in the past), *eventually*, or *sometime* in the future (*sometime* in the past), *Until* (*Since*), and *next* cycle (*previous* cycle).

Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real-time systems [4]. MTL extends LTL by supporting the specification of relative- and real-time constraints. All four LTL future-time operators (*Always*, *Eventually*, *Until*, *Next*) can be characterized by relative- and real-time constraints specifying the duration of the temporal operator. Hence, for example, the MTL assertion "Always $_{< 20}$ commandResult > 0", states that commandResult > 0 must hold every cycle until 20 cycles in the future.

MTL with time-series constraints (MTLS) enables the specification of requirements in which propositions include temporal instances of variables. Consider the following *automotive cruise control* code with a stability assertion (using embedded TemporalRover syntax [5]) requiring speed to be 95% stable while cruise is set and not changed:

```
void cruise(boolean cruiseSet, boolean cruiseChange,
            boolean cruiseOff, boolean cruiseIncr, int speed)
{
   /* Cruise Controller functionality */
    …
   /* TRBegin
         TRAssert {
            Always ( {cruiseSet} =>
               {speed*0.95 < speed' &&
                              speed' < speed*1.05}
               Until $speed$ {cruiseChange || cruiseOff} )
          } => {…} // user actions
      TREnd */
}
```

In the example *speed* is a temporal data variable, which is associated with the *Until* temporal operator. This association implies that every time the *Until* operator begins its evaluation, possibly in multiple instances (due to non-determinism), the speed value is sampled and preserved in the *speed* variable of this instance of the *Until*; this value is referred to as the *pivot* value for this *Until* node instance. Future speed values used by this particular evaluation of the *Until* statement are referred to using the *prime* notation, that is, as *speed'*; these future instances of the speed value are referred to as *primed values*. Hence, if the speed value was 100Km/h when *cruiseSet* is true, then the pivot value for speed is 100, while every subsequent speed is referred to as *speed'* and must be within 5% of the pivot speed value.

Note how speed is *declared* using the *$speed$* notation to be a temporal data variable associated with the *Until* operator. This declaration indicates to TemporalRover that it should be sampling a pivot value from the environment in the first cycle of the *Until* operators lifecycle, and to refer to all subsequent samples of speed as *speed'*.

Similarly, the following example consists of a *monotonicity* requirement for the cruise control system, where speed is monotonically increasing while the Cruise Increase (*cruiseIncr*) command is active:

```
TRAssert {
   Always ( {cruiseIncr} =>
        {speed <= speed' && (speed=speed') >= 0}
        Until $speed$ {!cruiseIncr} )
} =>  {…} // user actions
```

In this example the temporal data variable *speed* is sampled upon every *cruiseIncr* event, and is compared to the current value (*speed'*) every cycle. The latest speed value is then saved in the pivot for next cycle's comparison.

DBRover is an MTLS monitoring tool based on the TemporalRover code generator of [5]. It consists of a GUI for editing temporal assertions, an MTLS simulator, and an MTLS execution engine. DBRover builds and executes temporal rules for a target program or application. In run-time, DBRover listens for messages from the target application and evaluates corresponding temporal assertions. Hence, in the cruise-control example above, DBRover will listen for Boolean messages pertaining to the run-time values of the cruiseSet, cruiseChange, and cruiseOff Boolean propositions, as well as the run-time value of the speed variable. DBRover then evaluates the corresponding MTLS assertion for that cycle. Monitoring is performed *on-line*, namely, DBRover operates in tandem with the target program, and re-evaluates assertions every cycle. DBRover uses an underlying algorithm that does not store a history trace of the data it receives; it can therefore monitor very long, and potentially never ending, executions of target applications.

4. MISSILE DEFENSE SYSTEM – A CASE STUDY

In this section, we illustrate the formal specification and run-time verification of timing specifications with a hypothetical ballistic missile defense system (BMDS). The BMDS is an integrated system-of-systems which provides a layered defense that employs complementary sensors and weapons to engage threat targets by land, sea, air, or space in the boost, midcourse, and terminal phases of flight, and incrementally deploying that capability. In parallel, sensor suites, command and control, battle management and communication (C2BMC) will be developed to form the backbone of the BMDS.

4.1 Use Case Analysis

To understand the requirements and constraints of the proposed system, we developed six UML use cases to identify the external agents and systems that are involved in a typical missile-defense scenario and the necessary interactions between these entities:

1. Detect Potential Threat Ballistic Missile - The goal of this use case is to detect possible threat ballistic missiles, and push the track data onto the sensor net.
2. Generate and Transmit a Local Track - This is a sub-use case of 1. The goal of this use case is to have a sensor generate a local track based on valid detection parameters of the sensor.
3. Cooperatively Track and Classify Threat Ballistic Missiles - The goal of this use case is to identify and type-classify the threat ballistic missiles, develop fire-quality tracks for engagement solutions, and forward the target track list to weapons net.
4. Cooperative Weapons Assignment - The goal of this use case is to assign targets to weapons via cooperative target bidding.
5. Engage Targets - The goal of this use case is to engage threat ballistic missile.
6. Assess Kill - The goal of this use case is to determine the kill status of the threat ballistic missile.

Details of the use cases can be found in [10,12].

4.2 The UML-RT Architecture Models

The technological complexity, physically dispersed geography, and distributed nature of global ballistic missile defense necessitate a distributed approach to ballistic missile defense battle management. Based on the use cases, we developed the top level of a distributed architecture shown in Figure 4, along with the corresponding UML-RT model shown in Figure 5.

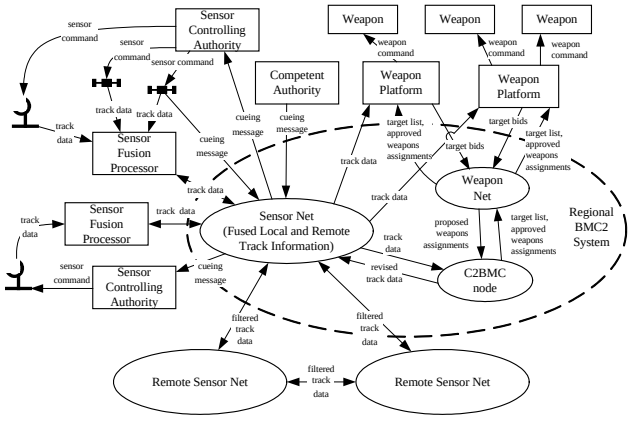

Figure 4. A distributed C2BMC architecture

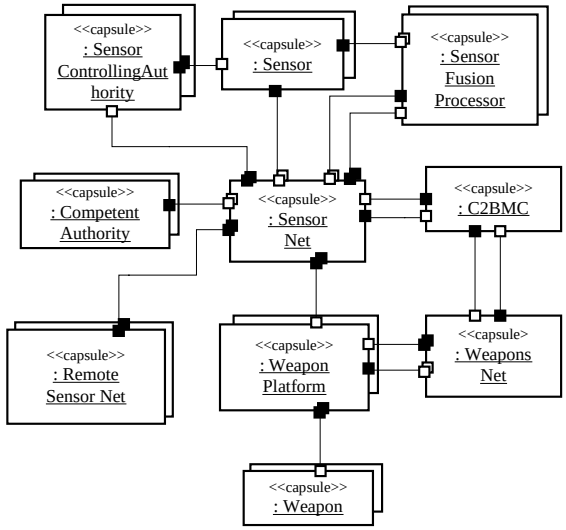

Figure 5. The UML-RT model for the C2BMC architecture

The overarching C2BMC System will consist of a loosely coupled set of regional C2BM systems; geographically separated networks interconnected much like the Internet. The intent is to allow all participants to pull the information from specific regions as desired, but also to ensure that time-critical information can be pushed to those geographically collocated units that need it to effect destruction of a threat missile or to hand-off the information to non-geographically collocated units as a missile transits from one region to another. Note that the various sensors and weapons may be connected to more than one regional C2BM system via proxy. The advantage is that geographic location is a "don't care" in that context. The real-time nature of the battle requires that all sensor information be local to fight the battle. As the missile continues in its flight, the real-time battle management, together with some of the sensors and weapons, will handover to another regional C2BM system. The use of the Broker pattern [1] will ease the handover of the assets from one region to another. By distributing the networks in this manner, information regarding any ballistic missile threat is available and accessible to all participants as desired, but will not overburden the network by having all the information presented to all units all the time; this will in theory provide increased availability of data, more localized control, and improved response times of the units to counter the threat.

Each unit (battle manager, sensor, weapons, etc.) connecting to a regional C2BM system publishes a unit profile that contains knowledge of the geographic location of the unit and its network address so that only data and information relevant to a particular unit (or region) is forwarded to that unit (or region). For example, fire-control data from another theater or region may not be useful and hence will stay local, while threat information from other theaters or regions will probably serve as situational awareness and be made available to other regions.

Each regional C2BM system consists of three major subsystems: a Sensor Net, a Weapons Net, and a C2BMC node, where the Sensor Net refers to a distributed system that provides the sharing of track data among Sensor Fusion Processors, Weapons Net, Weapon Platforms and the C2BMC node, the Weapons Net refers to a distributed system for cooperative target assignment, and the C2BMC node refers to automation support for the Command/Control, Battle Management and Communication (C2BMC) functions.

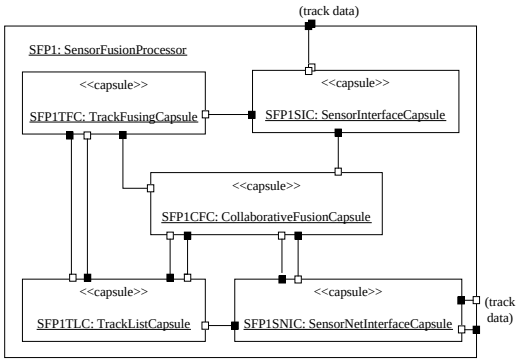

Figure 6. The internal structure of the Sensor Fusion Processor capsule

We refined the internal structures of the Sensor capsule, the Sensor Net capsule and the Sensor Fusion Processor capsule using the HIPO technique. Figures 6 shows the internal structure of the Sensor Fusion Processor (SFP) capsule, which consists of five sub-capsules (*Sensor Interface*, *Track Fusing*, *Collaborative Fusion*, *Track List* and *Sensor Net Interface*). The Sensor Interface capsule serves as the primary interface to all assigned sensors. It sends all tracks to the Track Fusing capsule if it is receiving data from more than one sensor; otherwise, it passes the data directly to the Collaborative Fusion capsule. The Track Fusing capsule takes multiple tracks per target from the Sensor Interface capsule, correlates or fuses them into one single track per target in real time. It also performs track discrimination as a backup to the sensor's native discrimination capability to prevent overload on the Sensor Net. The Collaborative Fusion capsule takes fused or raw local tracks (one per target) and fuses them with tracks received from other SFPs via the Sensor Net. The Track List capsule is responsible for compiling and providing the internal list of tracks for the SFP and preventing duplicates. It provides this data to both the Track Fusing capsule and the Collaborative Fusion capsule, and provides this information to other SFPs upon request via the sensor net. It also serves as a repository for commands received from the Sensor Net. The Sensor Net Interface capsule is responsible for pushing tracks from the Track List capsule to the Sensor Net and routing the incoming tracks from other SFPs via the Sensor Net to the Collaborative Fusion Capsule. (Readers can refer to [12] for the complete UML-RT models.)

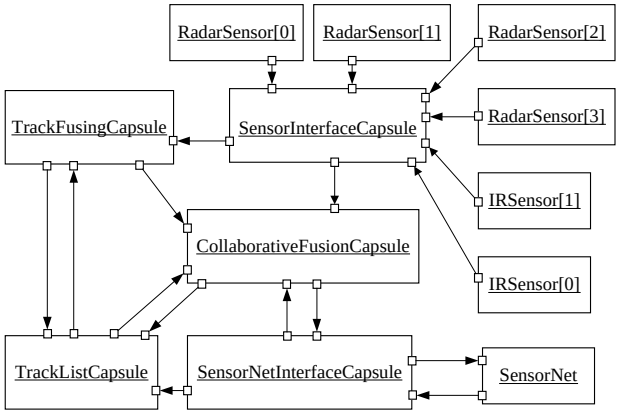

Figure 7. The OMNeT++ Model

We developed an OMNeT++ simulation shown in Figure 7 to identify potential bottlenecks at the Sensor Fusion Processor. It consists of twelve modules simulating the five sub-capsules of the Sensor Fusion Processor interfacing with two satellite Sensor capsules, four ground radar Sensor capsules and the Sensor Net capsule. The internal structures of these capsules are mapped to the C++ code of the corresponding modules. Figure 8 shows the graphical user interface for the simulation model shown in Figure 7.

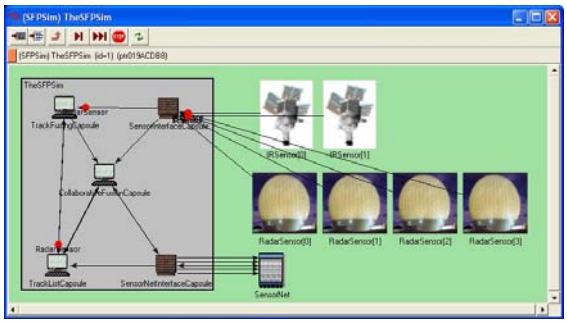

Figure 8. The Graphical User Interface

4.3 Formal Requirements Specifications and Run-time Verification of Temporal Assertions

Detection is critical to any ballistic missile defense system; the bottom line is that if one does not see the threat missile or know that it is coming then one cannot defend against it. The proposed BMDS relies on a network of sensors to detect, discriminate and track every ballistic missile event, and provide their parametric data on the contacts to the C2BMC to develop combined tracks through fusion and correlation in order to develop a weapons solution to prosecute the target. We can express many of these sensor requirements formally in terms of time-series temporal logic.

For example, the requirements that all sensor-tracks must be correlated and fused can be expressed formally, using the TemporalRover syntax, as follows:

```
/* TRBegin
      TRAssert For{sensorTrack1, sensorTrack2} {
         Always ({flightObjectFor(sensorTrack1) ==
                  flightObjectFor(sensorTrack2)} =>
            Eventually <nn1 (
                {fusedTrackOf(sensorTrack1) != null} &&
                Eventually<nn2 (
                    {fusedTrackOf(sensorTrack1).ID ==
                     fusedTrackOf(sensorTrack2).ID)} &&
                    monotonicity(
                              fusedTrackOf(sensorTrack1))
                )
            )
      TREnd */
```

where

- For{sensorTrack1, sensorTrack2} is like the logical "forall sensorTrack1, sensorTrack2", the monitor will perform separate evaluations for every pair of sensor tracks;
- fusedTrackOf(sensorTrack) provides a fused track for a sensor track, and fusedTrackOf(sensorTrack1) != null means that there exists a fused track for the sensorTrack;
- flightObjectFor(sensorTrack) means the actual missile flying out there that is sensed by a sensor and then captured as sensorTrack;
- Monotonicity( ) is a monotonically increasing requirement of some artifact (e.g., time with respect to a common reference); time series constraints, similar to the cruise control constraints of section 3 are used to specify this requirement;
- The constants nn1 and nn2 are temporal conditions (e.g., durations, number of trails) or Boolean functions that can be evaluated at run-time.

Note that although the monotonicity requirement seems trivial, it is less trivial when considering the fact that it is a requirement for monotonicity of a fused track, which derives its information from a plurality of sensor tracks. If, due to an error, the software confuses sensor tracks thereby updating fused-track information from the wrong sensor, then from time to time it may appear that the fused track is not advancing monotonically.

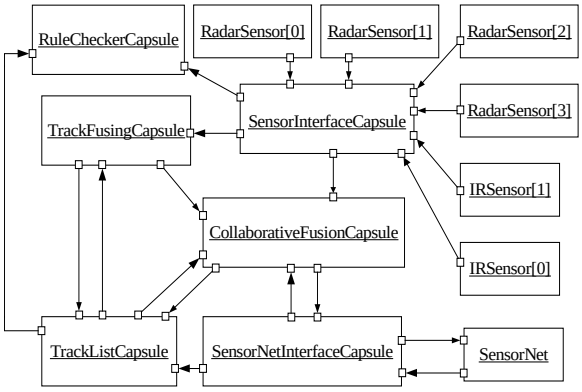

Figure 9. The Modified OMNeT++ Model

Figure 9 shows the modified OMNeT++ model with the additional *RuleCheckerCapsule* module. The new module receives raw and fused track data from the *SensorInterfaceCapsule* and the *TrackListCapsule* respectively, and executes the code snippets generated by DBRover to evaluate the Boolean conditions of the rule segments and sends the result to the DBRover Runtime Monitor for real-time temporal rule verification (Figure 10).

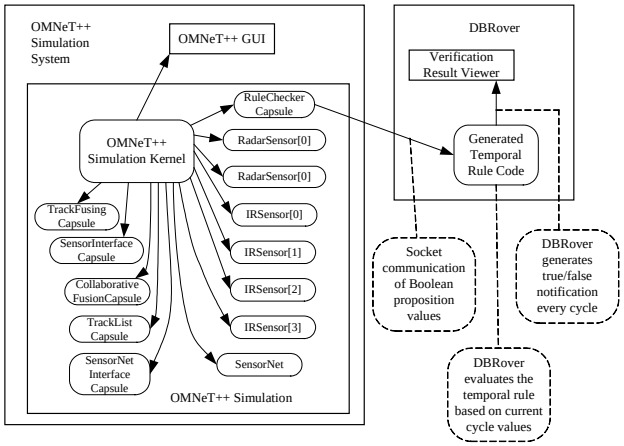

Figure 10. Architecture of the integrated OMNeT++ Simulator / DBRover Run-time Monitor System

5. DISCUSSIONS AND CONCLUSION

This paper shows that run-time monitoring and verification can be applied much earlier in the design process, in tandem with rapid prototyping to study the timing requirements of complex systems. The integration of the OMNeT++ simulation with the DBRover Run-time Monitor provides an effective way to check for the validity of the requirements and violation of the temporal assertions. Through the Use Case-Model-Simulation feedback cycle, we were able to identify potential bottlenecks in the architecture design, which led to redesign of some of its components. Revised temporal constraints are input to DBRover to generate executable timing specifications and new code snippets for re-instrumentation of the simulation code. Moreover, re-instrumentation of the simulation code is only needed if we modify the Boolean conditions of the rule segments. DBRover provides a special configuration, called the No Snippets Protocol, which allows the user to instrument the target code for a set of Boolean conditions in a one-time setup, after which changes to the rules on the DBRover side do not affect the client/target side.

6. ACKNOWLEDGEMENTS AND DISCLAIMER

The research reported in this article was funded by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

7. REFERENCES

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, **Pattern-Oriented Software Architecture: A System of Patterns**, New York: John Wiley & Sons, 1996.

[2] D.S. Caffall and J.B. Michael, "A New Paradigm for Requirements Specification and Analysis of System-of-Systems", **Lecture Notes in Computer Science, no. 2941 (Proc. Monterey Workshop 2002: Radical Innovations of Software and System Engineering in the Future)**, Berlin: Springer-Verlag, 2004, pp. 108-121.

[3] D.S. Caffall, **Conceptual Framework Approach for System-of-Systems Software Developments**, Master's thesis, Naval Postgraduate School, Monterey, Calif., Mar. 2003.

[4] E. Chang, A. Pnueli and Z. Manna, "Compositional Verification of Real-Time Systems", **Proc. 9th IEEE Symp. On Logic In Computer Science**, 1994, pp. 458-465.

[5] D. Drusinsky, "The Temporal Rover and ATG Rover", **Lecture Notes in Computer Science, no. 1885 (Proc. Spin2000 Workshop)**, Berlin: Springer-Verlag, 2000, pp. 323-329.

[6] B. Hailpern and S. Owicki, "Modular Verification of Communication Protocols", **IEEE Trans of Comm**., COM-31, No. 1, 1983, pp. 56-68.

[7] **HIPO – A Design Aid and Documentation Technique,** Report no. GC20-1851-0, IBM Corp., White Plains, N.Y., 1974.

[8] J. Liu, **Real-Time Systems**, Prentice Hall, 2000.

[9] Z. Manna and A. Pnueli, "Verification of Concurrent Programs: Temporal Proof Principles", **Lecture Notes in Computer Science, no. 131 (Proc. of the Workshop on Logics of Programs)**, Berlin: Springer-Verlag, 1981 pp. 200-252.

[10] J.B. Michael, P. Pace, M.T. Shing, M. Tummala, J. Babbitt, M. Miklaski and D. Weller, **Test and Evaluation of the Ballistic Missile Defense System: FY 03 Progress Report**, Tech. Report NPS-CS-03-007, Naval Postgraduate School, Monterey, Calif., Sept. 2003.

[11] J.B. Michael, M.T. Shing, J. Babbitt and M. Miklaski, "Modeling and Simulation of System-of-Systems Timing Constraints with UML-RT and OMNeT++", **Proc. 15th IEEE International Rapid System Prototyping Workshop**, Geneva, Switzerland, 28-30 June 2004.

[12] M.H. Miklaski and J.D. Babbitt, **A Methodology for Developing Timing Constraints for the Ballistic Missile Defense System**, Master's thesis, Naval Postgraduate School, Monterey, Calif., Dec. 2003.

[13] Object Management Group. **OMG Unified Modeling Language (UML) Specification 1.5**, March 2003. http://www.omg.org/technology/documents/formal/uml.htm

[14] A. Pnueli, "The Temporal Logic of Programs", **Proc. 18th IEEE Symp. on Foundations of Computer Science**, 1977, pp. 46-57.

[15] B. Selic and J. Rumbaugh, **Using UML for Modeling Complex Real-Time Systems**, Unpublished white paper, Apr. 4, 1998, http://www.rational.com/media/whitepapers/umlrt.pdf

[16] B. Selic, G. Gullekson and P. Ward, **Real-Time Object Oriented modeling,** New York: John Wiley & Sons, 1994.

[17] A. Varga, **OMNeT++ Discrete Simulation System (Version 2.3) User Manual**, Technical University of Budapest, Dept. of Telecommunications (BME-HIT), Hungary, Mar. 2002.